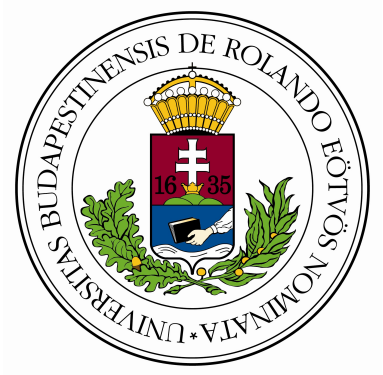
Disztributív és párhuzamos problémák megoldása - módszertanok, adatszerkezetek és algoritmusok

Tátrai Antal

Doktori értekezés tézisei

Eötvös Loránd Tudományegyetem
Komputeralgebra Tanszék



Témavezető: Dr. Kovács Attila

Informatika Doktori Iskola
Dr. Benczúr András D.Sc.

Az informatika alapjai és módszertana doktori program
Dr. Demetrovics János
a Magyar Tudományos Akadémia tagja

2012

Bevezetés

A többprocesszoros számítógépek és mobil eszközök korát éljük. Az iparban teljesen megszokott a több száz vagy ezer processzormagot tartalmazó hardver, de valószínűleg sokak otthoni számítógépében is legalább két processzor található. Szoftver oldalról közelítve azonban számos problémának csak szekvenciális algoritmikus megoldása ismert, amelyek alig, vagy egyáltalán nem tudják kiaknázni a párhuzamos környezet kínálta lehetőségeket. Talán a számítástudomány egyik legnagyobb kihívása, hogy biztonságos és jól skálázható párhuzamos algoritmusokat és adatszerkezeteket alkosson.
A disszertációmban elemeztem és megoldottam három olyan problémát, amelynek előtte vagy csak szekvenciális megoldása volt ismert, vagy pedig az ismert párhuzamos megoldás valamilyen szempontból javítható volt.

1.1. Erlang rendszerek párhuzamos indítása

Egy disztributív rendszer olyan processzorok halmaza, amelyek nem feltétlenül osztják meg a memóriát egymással és nincs közös órajel generátoruk, ellenben minden processzornak van saját lokális memóriája. A processzorok rendszerint valamilyen hálózaton keresztül kapcsolódnak egymáshoz és a kommunikáció üzenetküldéssel valósul meg közöttük.

Az Erlang egy disztributív programozási nyelv, amit az Ericsson fejlesztett ki azzal a céllal, hogy a rendszereihez legyen egy hibatűrő, párhuzamos, funkcionális programozási nyelve. Az Erlang egy ingyenes nyílt forráskódú fejlesztői környezet, aminek disztributív kernelje van. Az Erlang nyelvhez egy gazdag fejlesztői könyvtár is kapcsolódik, az Open Telecom Platform (OTP). Az OTP nemcsak egy egyszerű könyvtár, hanem a nyelvhez egy magas szintű folyamatmodellt definiál, valamint számos segédprogramot kínál, amelyek egyszerűbbé teszik a fejlesztést. Természetesen az OTP tervezési mintákat is tartalmaz, amiket az Erlang *behaviour*-őknek hív. A fejlesztői könyvtár a behaviour-ökkel együtt kiváló alapot nyújt robusztus, hibatűrő és gyors rendszerek készítéséhez. Az Erlang programok függvényekből állnak, amiket modulokban tárolunk. A függvényeket tetszőleges folyamat végrehajthatja és a folyamatok nyelvi szintű parancsokkal kommunikálhatnak egymással, akár aszinkron módon is. Egy folyamat létrehozása kevés processzoridőt és memóriát igényel. Az Erlang/OTP alapvető folyamatkezelési koncepciója a *supervision tree*-ken (felügyelő fa) alapszik, ahol minden csúcs egy különálló folyamat. A programjaink folyamatait „master-slave” viszonyba tudjuk állítani a supervisor node-ok (csomópontok) segítségével. Az Erlang program így egy folyamatfát (erdőt) alkot. A fa levelei oldják meg a végrehajtandó feladatot, a belső csúcsok pedig csak koordinálják a gyerekeik működését. Egy supervisor (belső) csúcs érzékeli és lekezeli ha egy gyereke „elszállt”.

Azoknak a folyamatoknak, akik egy folyamatfában szerepelnek, bizonyos megszorításoknak kell eleget tenniük. A legfontosabb talán az, hogy logikailag úgy kell felépülniük, hogy kezdetben valamilyen inicializálást végeznek el és csak utána állnak neki a tényleges feladatukat ellátni. Természetesen az inicializálás lehet üres is, de a kód struktúrájában ennek látszódnia kell, mivel az inicializáció lefutása után már teljesen elindult folyamatként fog működni.

Egy Erlang/OTP rendszer indulása a következőképpen történik: először elindul egy alrendszer, ami rövid úton elindít egy folyamatfát. Egy folyamatfa belső csúcsa először elindítja a bal szélső gyereket, majd megvárja amíg az elindult. Itt a szülő azt várja meg, hogy a gyereke befejezze az inicializációs részt a kódjából. Ennek befejeztét egy ACK üzenetküldés jelzi. Miután az ACK megérkezett, a belső csúcs indítja a következő gyerekét. Azaz egy ilyen folyamatfa szekvenciálisan indul.

Ezen túlmenőleg a folyamatfa szekvenciális indítása valójában egy sorrendet is definiál a folyamatok között. A probléma egy párhuzamos megoldása csak úgy képzelhető el, hogy definiálva van valamilyen alternatív sorrend a párhuzamosan induló csúcsok között.
A feladat:

Az Erlang rendszerek párhuzamos indításához a következő feltételeket kell betartani:

- A folyamatfákat nem szabad szétvágni, azaz nem megoldás az a párhuzamos indítás, amelyik után egyes folyamatok kénytelenek szülő nélkül magukban futni.
- A startnak biztonságosnak és gyorsnak kell lennie (gyorsabbnak mint a szekvenciális)
- Csak apróbb változtatások megengedettek a standard könyvtárakon.

1.2. Szoftveres gyorsítótárak tisztítása

A modern számítógép rendszerek gyakran használnak gyorsítótárakat azért, hogy a távoli lassú adatokat gyorsabban érjék el. Talán a legtipikusabb ilyen szolgáltatás a Domain név feloldás (DNS). Általában a név-cím feloldásokra a választ nem egy távoli szervertől kapjuk, hanem egy közelitől, aki eltárolta már az általunk kért név IP címét.
Mivel ezek az adatok valójában csak másolatai az eredeti adatoknak, rendszeresen szükséges ezek ellenőrzése, szinkronizálása. Minden adathoz tartozik egy szavatosság, aminek lejárta után az adat nem érvényes tovább (Time-To-Live, TTL). Általában az adatok érvényességének ellenőrzését egy speciális szál valósítja meg, amelyik egyszerűen végignézi az elemeket a gyorsítótárban és törli a már lejártakat.
A gyorsítótár egy absztrakt adattípus, ami többféleképpen megvalósítható. Általában fák vagy hash-táblák képezik az implementációk alapját. Én a hash-tábla alapú meg-

oldásokra koncentráltam. Párhuzamos környezetben számos szál próbálja meg elérni a gyorsítótárat, amelyeket szinkronizálni kell.

Általánosságban elmondhatjuk, hogy a tisztító szálnak végig kell néznie az egész adatszerkezetet. Nyilvánvaló, hogy ha tudná a szál, hogy mely elemek jártak le, sokkal gyorsabban befejezhetné a lejárt elemek karbantartását.
A feladat:

- Olyan adatszerkezetek készítése, amelyek gyorsabbá teszik a tisztítószál működését úgy, hogy a keresés sebessége nem lassul.

1.3. $\mathbb{Z}^n$ vektortér elemeinek osztályozása általánosított számrendszerek segítségével

Legyen $M \in \mathbb{Z}^{n\times n}$ mátrix a bázis és $0 \in D \subseteq \mathbb{Z}^n$ egy véges halmaz, amit a jegyek halmazának hívunk. Az (M, D) párt általánosított számrendszernek hívjuk, ha minden $x \in \mathbb{Z}^n$ pontra létezik egy $k \in \mathbb{N}$ egész szám, amire $x = \sum_{i=0}^{k} M^i d_{i_j}$, ahol $d_{i_j} \in D$. Természetesen ahhoz, hogy egy (M, D) pár általánosított számrendszer legyen, néhány feltételnek teljesülnie kell: az M mátrixnak expanzívnak kell lennie, a D halmaznak teljes maradékrendszert kell alkotnia modulo M, valamint $|M - I| \neq \pm 1$.

Definiáljuk a ϕ függvényt a következő módon: $\phi : \mathbb{Z}^n \to \mathbb{Z}^n$, $\phi(x) = M^{-1}(x - d)$, ahol $d \in D$ és $x \equiv d \pmod{M}$. Ha egy x pontból kiindulva a ϕ függvényt ismétlődve alkalmazzuk, akkor az így kapott pontok halmaza az x pont pályája. Egy periodikus pont pályáját ciklikusnak nevezzük. Mivel M expanzív, ezért létezik olyan norma, amire $\|M^{-1}\| < 1$. Definiáljuk S_k halmazokat a következőképpen: $S_k = \{x \in \mathbb{Z}^n | \ \|x\| < k\}$ ahol $k \in \mathbb{R}$. Mivel D véges halmaz, létezik egy olyan $C \in \mathbb{R}$ konstans, hogy minden x pont pályája egy idő után belekerül S_C halmazba és innentől kezdve benne is marad. Ezen állítás fényében az általánosított számrendszer tulajdonság átfogalmazható: Egy (M, D) pár akkor és csak akkor általánosított számrendszer, ha minden $x \in \mathbb{Z}^n$ pont pályája eléri a 0-át. Két egymástól egy kicsit eltérő problémát tudunk megfogalmazni az általánosított számrendszerekkel kapcsolatban. Eldöntési problémának hívjuk azt, amikor el kell dönteni egy (M, D) párról, hogy általánosított számrendszer-e, osztályozási problémának hívjuk azt a feladatot, amikor egy adott (M, D) pár minden ciklusát meg kell keresni.

Mindkét feladat osztályra ismert szekvenciális megoldás, azonban sajnos, amikor M sajátértékei közel vannak az egyhez, akkor S_C nagyon nagy lehet. Magasabb dimenziókban a bináris jegyhalmazt tartalmazó (M, D) párok (amikor $\det M = \pm 2$) általában ilyenek.

A feladat:

- Olyan párhuzamos algoritmusok konstruálása, amelyek hatékonyan oldják meg az eldöntés és az osztályozás feladatát.

Tézisek

2.1. Erlang rendszerek párhuzamos indítása

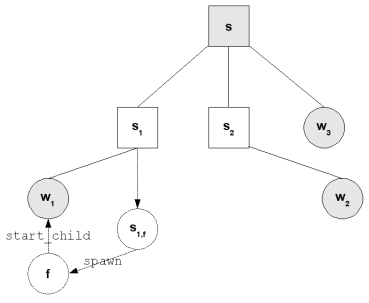
1. Tézis

A párhuzamos indításra olyan eljárást fejlesztettem, ami a folyamatfa gyerek csúcsainak párhuzamos indításán alapszik oly módon, hogy közben nem rontja el a folyamatfa gyerekmonitorozási tulajdonságát, valamint a kapott módszer funkcionálisan teljesen kompatibilis a már meglévő szekvenciálissal és csak kevés változtatást igényel az `stdlib` *könyvtárban.*

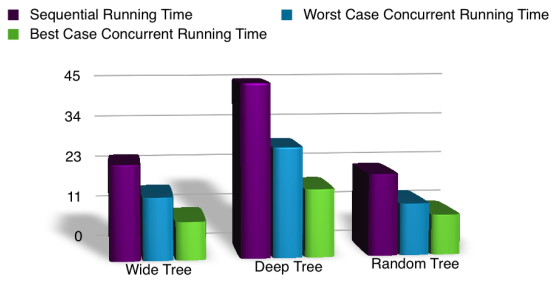
Az általunk felvázolt megoldás legjobban talán az Apple MacOS X rendszerében használt indító-scriptekre hasonlít [5], az eredeti Erlang indítási séma pedig a következő publikációkból ismerhető meg: [6, 7, 8, 9, 22].

Definiáltam a *feltétel* (condition) fogalmát. Kezdetben minden feltétel hamis. Előfeltételnek hívom azokat a feltételeket, amik egy folyamat elindításához szükségesek. Egy folyamat csak akkor indulhat el, ha minden előfeltétele igaz. Ha egy folyamat elindult (lefutott az inicializációs része), akkor a hozzá tartozó feltételt igazra állítja. Egy folyamathoz egy feltétel tartozhat legfeljebb. Így egy gráfot kapunk, ami rendszerint nem összefüggő és a különböző folyamatok egymáshoz viszonyított indulási sorrendjét adja meg.

A párhuzamos indításhoz szükség volt egy Erlang trükkre, amit a 2.1. ábra mutat. Azért, hogy párhuzamosan tudjunk indítani több folyamatot, azt kell elérni, hogy az



ábra 2.1: *Az ábrán látszódik, hogy egy dummy supervisor csúcs lesz beszúrva (1) a belső csúcs alá, hogy megőrizze a folyamatfa gyerekmonitorozási tulajdonságát és végrehajtsa a párhuzamos indítást. A supervisor folyamatok négyzetekkel vannak reprezentálva. A megmaradó folyamatoknak folytonos szegélyük van, a temporálisoknak pedig szaggatott. A fa közepén a már futó folyamatok láthatók, miután a dummy folyamatok elhaltak (s_2, w_2).*



ábra 2.2: *Különböző karakterisztikájú folyamatfákon mért szekvenciális és párhuzamos indítási sebességek.*

inicializáló szakaszát egyszerre két (vagy több) folyamat tudja futtatni. Mivel az ACK üzenetek miatt nem lehet ezt egyszerűen megoldani, valahogy át kell verni a rendszert, és ACK üzenetet kell generálni. Ha megnézzük az ábrát, akkor láthatjuk rajta, hogy az s folyamat szeretné elindítani a w_1-es és a w_2-es folyamatait párhuzamosan. Ekkor a következő történik. s ahelyett, hogy elindítaná w_1-et, elindítja s_1-et, ami egy üres (dummy) supervisor folyamat. Ez annyit tesz, hogy az inicializáló részében csak egy gyerek indítása van beletéve ($s_{1,f}$), ami viszont az inicializáló részében nem csinál semmit. Következés képpen s_1 azonnal visszaküldi az ACK-t s-nek, aki elkezdi indítani w_2-t. $s_{1,f}$ lényegében csak aszinkron módon elindít egy másik folyamatot (f-et) és azután terminál. De s_1 ettől még nem terminál, hanem egy olyan supervisor folyamat lesz, amelyiknek nincs gyereke. f nincs beláncolva egy folyamatfába se, tehát rá nem kell várnia senkinek. f Meghívja s_1-nek a `start_child` függvényét és elindítja w_1-et. Mivel a `start_child` függvény blokkolódik addig, amíg w_1 nem hajtja végre az inicializáló részt, ezért f szintén blokkolódik, de mivel rá nem vár senki, ez nem számít. Ezután f terminál és s_1-en keresztül w_1 is be lesz fűzve a folyamatfába, de ez már teljesen párhuzamosan történik w_2 indításával (sőt w_3-éval is).

Természetesen egy-egy folyamatnak az elindulásakor ellenőriznie kell, hogy minden előfeltétele teljesült-e. Ezt az `stdlib`-be írt változtatások automatikusan megteszik és blokkolják az inicializációs szakasz futását, amíg minden előfeltétel igaz nem lesz. Így előfordulhat, hogy például w_1 nem indulhat még, mert egy másik folyamat nem indul még el, de ettől w_2 indulhat, ha minden előfeltétele teljesül.

A fenti módosításokat elkészítettem és implementáltam, valamint a kapott eredményeket kiértékeltem. Ez látható a 2.2. ábrán.

2.2. Szoftveres gyorsítótárak tisztítása

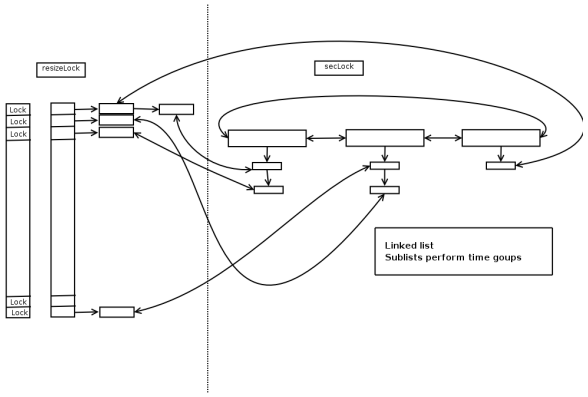
2. Tézis

Másodlagos adatszerkezeteket fejlesztettem a hash-tábla implementáció mellé, amik csak a mutatókat és a lejárati időket tartalmaznak. A tisztítószál a másodlagos adatszerkezetek segítségével gyorsabban meg tudja keresni a lejárt elemeket. A másodlagos adatszerkezetek

nem szükségesek a kereséshez, így a keresések aszimptotikus futásideje nem változott.

Az 1980-as években Carla Sch. Ellis több sokzáras hash-táblát publikált [11,12], amik hatékonyabbak voltak az egyszerű egyzáras megoldásoknál. 2002-ben Maged M. Michael egy zár nélküli atomi compare-and-set műveleten alapuló hash-táblát publikált [13,14,15, 16, 19, 20]. Ennek a hash-táblának a sebessége megegyezett az Ellis féle zár alapú hash-tábláéval, amikor ugyanannyi folyamat volt mint processzor. Azonban amikor sokkal több folyamat volt mint processzor, Michael hash-táblája sokkal gyorsabb maradt. Michael hash-tábláját azonban nem lehetett átméretezni, csak a listák hossza nőtt benne. Később Nir Shavit és Ori Shalev készített egy átméretezhető és atomi műveletekkel működő hash-tábla implementációt [21]. A kutatásomhoz Ellis hash-tábláját használtam föl.

Több fajta másodlagos adatszerkezetet is készítettem. Az első vizsgált adatszerkezet a bináris kupac volt. Az a probléma ezzel az adatszerkezettel, hogy egy tisztítószál rendszerint több elemet szeretne egyszerre törölni, így egymás után több elemet kell törölni a kupacból is. Ennek az aszimptotikus futásideje darabonként $O(log(n))$. Természetesen a beszúrásoknak is hasonló gépigénye van. Kipróbáltam a bináris fákat is, de ott is hasonló problémába ütköztem.
Végül egy speciális adatszerkezetet készítettem, amit *bucket-list*-nek neveztem el. (A 2.3. ábra ezt mutatja.) Az alapötlet az volt, hogy az időt felosztom akkora szeletekre, amekkorát egy tisztítószál alszik két lefutása között. Minden egyes időpillanat pontosan egy ilyen szeletbe tartozik bele. Amikor beszúrunk egy új elemet, akkor megkeressük azt a szeletet, ami azt az időpillanatot tartalmazza amikor lejár az elem, és oda beszúrunk egy mutatót az adatra. Nyilván azok a szeletek, amikor nem jár le adat, nincsenek tárolva. Amikor felébred a tisztító szál, akkor semmi más dolga nincs, mint kitörölni (vagy frissíteni) azokat az elemeket, amik az utolsó szeletben vannak felsorolva. Ha feltesszük, hogy konstans TTL értékekkel dolgozunk, akkor pedig a beszúrás is gyors lesz, mivel mindig a lista másik végébe akarunk beszúrni. Aszimptotikusan kifejezve ez azt jelenti, hogy



2.3. ábra: *Láncolt hash-tábla bucket-list másodlagos adatszerkezettel*

egy beszúráshoz általában $O(ts)$ műveletigény szükséges, ahol ts a létező szeletek száma, azonban konstans TTL mellett ez $O(1)$. A törölt elemek megkeresése pedig mindig $\Theta(ee)$ időt vesz igénybe, ahol ee a törlendő elemek száma.

Implementáltam és lemértem a fent leírtakat. Sajnos a zárak platformfüggősége miatt értelmetlen mások eredményeivel összehasonlítanom az eredményeket. Eredmény, hogy sebességnövekedést találtam egy egyszerű gyorsítótár és a javított változatok között, de csak nagyon kis szeletek mellett.

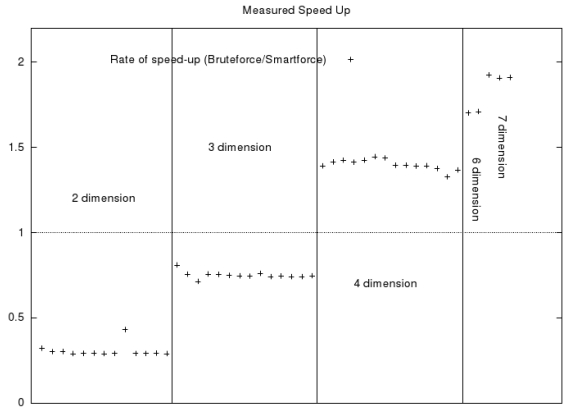
2.3. $\mathbb{Z}^n$ vektortér osztályozása általánosított számrendszerek segítségével

3.1. Tézis

Olyan master-slave modellen alapuló megoldásokat fejlesztettem, amelyek egy GRID cluster-en meg tudják oldani mind az osztályozás, mind pedig az eldöntés problémáját.

Ismert, hogy (M, D) minden periodikus eleme a $-H$ zárt halmazban van. A H halmaz 0 egészrészű számok halmaza [17]. Létezik egy olyan módszer, amely egy n-dimenziós kockába zárja a H halmazt [17]. A korábbi (szekvenciális) algoritmus a határoló n-dimenziós négyzet pontjainak ciklikusságát ellenőrizte le. Az első GRID-es megoldásom az n-dimenziós téglalapot szeletekre (slice) vágta. A master algoritmus elvégezte a vágást és a szeletek leírását elküldte a slave gépeknek. Nyilvánvaló, hogy ha valahol a ϕ függvény olyan pontba érkezik, amelyet már leellenőriztünk, akkor nem végezzük el a további iterációkat. Az első GRID algoritmus a szeletek készítéséhez a visszalépéses technikát alkalmazta. Mivel ezt a slave gépek is tudták, így elég volt csak egy kezdőpontot és a szelet hosszát átküldeni. A már biztosan ellenőrzött pontok halmazát pedig egy nem túl bonyolult függvénnyel lehet számolni. Elég csak azt megnézni, hogy a felsorolásban vajon előrébb van-e a pont, mint a kezdeti pont. Ha a ponton végzett ϕ iteráció egy olyan pontba érkezik, amelyik a felsorolásban előbb szerepel mint az eredeti pont, akkor biztosan megállhatunk, mert innentől már leellenőriztük a pontokat. Egy 3-dimenziós hasáb visszalépéses bejárása a pontokat hasonló módon sorolja fel, mintha vizet öntenénk egy kockába. Mivel azonban a ϕ függvény a 0 pont felé iterál ezért szerencsésebb lenne egy olyan térfelosztás, ami az n-dimenziós téglalap közepétől haladna kifelé. Az előző algoritmus javítható egy másfajta térfelosztással. Az új térfelosztás a *téglákon* (bricks) alapszik, amik egyszerű n-dimenziós téglalapok, csak a méretük kicsi. Néhány téglát együtt *hasábnak* (prism) nevezünk, ha együtt egy összefüggő n-dimenziós téglalapot alkotnak. A master spirális sorrendben generálja a téglákat. Egy slave egyszerre mindig egy téglát ellenőriz le, de megkapja azt a hasábot is, amelyik határos a téglával amit

ellenőriz. Ez az utóbbi azért kell, hogy a már biztosan ellenőrzött pontok halmazát meg tudjuk határozni. (Azaz azok a pontok amelyek egy hasábban vannak, már biztosan le vannak ellenőrizve.) Mivel az első tégla körbeöleli a 0-át és utána körkörösen generáljuk a téglákat, ezért a ϕ függvény iterációi gyorsabban be tudnak érni olyan pontba, amit már biztosan leellenőriztünk. A 2.4. ábra a két algoritmus futásidejének összehasonlítását mutatja be.



2.4. ábra: *A két gridalgoritmus egymáshoz viszonyított futásiejei pár kis dimenziós esetben*

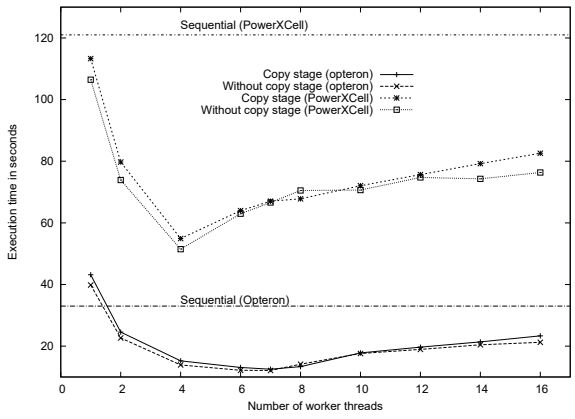
3.3. Tézis

Párhuzamos megoldásokat készítettem a Brunotte algoritmusra, amit két platform lehetőségeire optimalizáltam.

A Brunotte algoritmus [10, 18] szintén a ϕ függvény segítségével oldja meg az eldöntés és az osztályozás problémáját, de ellenben az eddigi megoldásokkal, egy kezdetben csak pár elemből álló halmazt bővít és az új elemekre végzi el az iterációt. A bővítés véges és ha az algoritmus nem talál periodikus pontot (a 0-án kívül), akkor az (M, D) pár GNS. Alapvetően két párhuzamos megoldást készítettem a problémára. Az első egy általános többprocesszoros, osztott memóriás számítógépen fut, a másik pedig az IBM által készített Cell Broadband Engien architektúrán. Mind a két megoldáshoz sok elemzés társult, hogy a különböző paramétereket tudjam vizsgálni. Az összehasonlítás megmutatta, hogy (természetesen) mind a két megoldás gyorsabb volt mint a szekvenciális, de a Cell architektúrán működő program szinte mindig lassabb volt az általános változatnál. Az egyik összehasonlítás eredménye látható a 2.5. ábrán.

A szerző publikációi

[1] Bánsági, A., Észiás, B. G., Kovács, A., and Tátrai, A. Source code scanners in software quality management and connections to international standards. *Annales Universitatis Scientarium Budapestinensis, Sectio Computatorica 37* (2012), 81 –93.



2.5. ábra: *Az egyszerű osztott memóriás környezetben és a CBEA architektúrán futó Brunotte algoritmusok futási eredményei*

[2] Kovács, A., Burcsi, P., and Tátrai, A. Start-phase control of distributed systems written in Erlang/OTP. *Acta Universitatis Sapientiae Informatica 2*, 1 (2010), 10 – 28.

[3] Tátrai, A. Parallel implementations of Brunotte's algorithm. *Journal of Parallel and Distributed Computing 71*, 4 (2011), 565 – 572.

[4] Tátrai, A., Dezső, B., and Fekete, I. Concurrent implementation of caches. In *7th International Conference on Applied Informatics* (2007), vol. 1, pp. 361 – 374.

Irodalomjegyzék

[5] Mac OS X Startup Items `http://developer.apple.com/documentation/MacOSX/Conceptual/BPSystemStartup/BPSystemStartup.pdf` .

[6] Open Source Erlang `www.erlang.org`.

[7] Armstrong, J. *Making reliable distributed systems in the presence of errors*. PhD thesis, Royal Institute of Technology, Stockholm, 2003.

[8] Armstrong, J. L., and Virding, R. One pass real-time generational mark-sweep garbage collection. In *IWMM '95: Proceedings of the International Workshop on Memory Management* (London, UK, 1995), Springer-Verlag, pp. 313–322.

[9] Blau, S., and Rooth, J. AXD 301 - A New Generation ATM Switching System, 1998.

[10] Brunotte, H. On trinomial bases of radix representations of algebraic integers. *Acta Scientiarum Mathematicarum* (2001).

[11] Ellis, C. S. Extendible hashing for concurrent operations and distributed data. In *PODS '83: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1983), ACM, pp. 106–116.

[12] Ellis, C. S. Concurrency and linear hashing. In *PODS '85: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1985), ACM, pp. 1–7.

[13] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), 5.

[14] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing* (London, UK, 2001), Springer-Verlag, pp. 300–314.

[15] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[16] Iii, W. N. S., and Scott, M. L. Nonblocking concurrent data structures with condition synchronization. In *In Proc. of the Intl. Symp. on Distributed Computing (DISC* (2004), pp. 174–187.

[17] Kovács, A. On computation of attractors for invertible expanding linear operators in $\mathbb{Z}^k$. *Publ. Math. Debrecen 56/1-2* (2000), 97–120.

[18] Kovács, A., Burcsi, P., and Papp-Varga, Z. Decision and classification algorithms for generalized number systems. *Annales Univ. Sci. Budapest, Sect. Comp. 28* (2008), 141–156.

[19] Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (New York, NY, USA, 2002), ACM, pp. 73–82.

[20] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing* (New York, NY, USA, 2002), ACM, pp. 21–30.

[21] Shalev, O., and Shavit, N. Split-ordered lists: Lock-free extensible hash tables. *J. ACM 53*, 3 (2006), 379–405.

[22] Virding, R., Wikström, C., and Williams, M. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.